Inter-Office Memorandum

To	Mesa Users	Date	October 17, 1977
From	Ed Satterthwaite	Location	Palo Alto
Subject	Mesa 3.0 Compiler Update Mesa Language Changes	Organization	CSL

XEROX

Filed on: [MAXC]<MESA-DOC>MESACOMPILER30.BRAVO

This release of the Mesa compiler introduces several changes of interest or importance to all Mesa programmers. A number of changes in the language have been made. Some of these will cause programs acceptable to previous compilers to be rejected unless those programs are modified. The reasons for making such changes are the following:

To support new ways of describing configurations and binding them together.

To add new facilities to the language.

To prepare for certain language extensions that are now fairly well understood and for which preliminary designs exist.

Language Changes Related to the Binder

Overview

The following overview might be helpful in understanding the changes related to the new binder. See the release summary for further references.

The binder produces a *configuration*, which is a collection of module instances. Each module instance is represented by a global frame. Resolution of intermodular references is based upon copying descriptor and pointer values from well-defined *interfaces* into these global frames. More precisely, the binder produces *configuration descriptions*, which must be "relocated" by the loader to produce an actual configuration.

There are two kinds of modules: DEFINITIONS and PROGRAM. The text of either kind implicitly defines a type. In the case of a program module *X*, this is a *frame type*, denoted by FRAME[*X*]. Values of this type are created in the (frame) heap, either by loading or by the NEW operation. They cannot be embedded within larger aggregates but are referenced indirectly through pointers. In the case of a DEFINITIONS module, the sequence of declarations implicitly defines an *interface type*. There is no explicit name for this type, but it is much like a record type with a field for each item in the interface. Instances of interface types, called *interface records*, can appear only as components of global frames, where they are anonymous.

A program can *export* an interface, in which case a (partially) initialized interface record is created by the compiler. The initializing values represent procedures, signals, etc., declared within the program. (These values are "relocated" and made instance-specific each time the program is instantiated.) A program can also *import* an interface to gain access to externally defined procedures and the like. In this case, the interface record is left uninitialized by the

compiler. One job of the binder is to merge exported instances of each particular interface type and to assign the result to imported interfaces of the same type. The binder's Configuration Description Language is used to specify and control these assignments.

A program also exports itself (as a pointer to its global frame) and can import instances of other programs (again, with access through frame pointers). In this case, the binder's job is to locate and assign the required (relocatable) frame pointers.

Note

The preceding discussion is conceptually accurate but should not be taken literally. The actual implementation of interfaces is somewhat more complicated and much more space-efficient than implied here. Exported interface records are part of each object file but have no existence during execution; furthermore, only those fields of imported records that are actually referenced occupy space in the global frame. Since the interface records do not exist as such during execution, they are sometimes called *virtual interface records.*

Because the binder is a preprocessor, any code required to compute and assign the values of initialized variables cannot be run during binding. For uniformity, the language definition has been changed so that the effect of the NEW operator is limited to creating an instance of a global frame. The frame must be STARTed to pass any required parameters and to initialize any nonconstant variables. The binder and loader perform the equivalent of a NEW but not a START.

Defining Interfaces

An interface type is defined by a DEFINITIONS module. The form of such a module has not changed. It contains two sorts of declarations:

(1) Constant definitions (including type definitions)
(2) Interface element definitions (procedures, signals, etc).

By convention, items of the first sort have identical values in all instances of the interface and can be referenced by specifying just the interface type. The fields of an interface record contain values of the second sort and correspond to the so-called "externals" found in previous versions of Mesa.

One new type of interface element is available. The type constructor

```
ProgramTC ::= PROGRAM ParameterList ReturnsClause
```

defines a type that can be used to declare a *program variable.* As part of an interface record, the value of a program variable is a pointer to a global frame of a like-named program. The declaration of a program variable specifies only input/output types; it does not provide access to the internal structure of a particular global frame. Uses of program variables are discussed below.

Defining Program Modules

A module gains access to another by including the (compiled) definitions file. As before, the DIRECTORY construct makes the connection between the Mesa identifier denoting the module and the name of the object file. Note, however, that identifiers introduced in the DIRECTORY now *must* match the module identifiers appearing in the original source.

Imported and exported items are "declared" in the heading of a PROGRAM module using the following syntax (cf. Appendix D, Mesa Language Manual):

```
ModuleHead ::=    DEFINITIONS ShareList
           |      PROGRAM ParameterList ReturnsClause
                      ImportList ExportList ShareList
ImportList ::=    empty | IMPORTS ModuleList
ExportList ::=    empty | EXPORTS IdList
ShareList  ::=    empty | SHARES IdList
ModuleList ::=    ModuleItem | ModuleList , ModuleItem
ModuleItem ::=    identifier | identifier : identifier
```

> The symbol DATA can be used in place of PROGRAM, but there is no longer a distinction between PROGRAM and DATA modules. Note also that the previous concept of IMPLEMENTING has been replaced by EXPORTS, and SHARES replaces SHARING.

Exporting Interfaces

The value of each identifier in the EXPORTS list must be an interface type, i.e., a DEFINITIONS module named in the directory.

Procedures, signals, and errors are exported if they are public, have constant initialization and are named in some exported interface. In addition, the program itself (in the form of its global frame) is exported as part of an interface if its identifier appears there with an appropriate PROGRAM type. (The global frame can also be exported independently of any interface; see below.) The compiler checks that the type of each exported item is assignment compatible with the type of the corresponding interface item. An item can be exported through more than one interface.

It is permissible for a module to both import and export an interface; this is the normal case when a number of modules cooperate to provide a single interface.

> If a module exports *Defs* and defines a public identifier *id*, then *Defs.id* is bound directly (by the compiler) to the local definition, e.g., local procedure calls are used.

Exporting an interface does not automatically provide access to its private components. Specifying SHARES (formerly SHARING) allows such access to any module but does not automatically imply EXPORTS.

Importing Interfaces

After a module has been loaded, its imported interface records contain the linkages to other modules in the configuration. References to such linkages take the usual forms. The identifier of an interface item (*Item*) can be qualified by the name of the interface record (*Defs.Item*[...]); alternatively, an interface record can be OPENed and the corresponding identifiers used without qualification (*Item*[...]).

Arbitrary identifiers (preceding ":" in the IMPORTS list) can be associated with imported interface records so that several instances of the same interface type (perhaps bound differently) can be distinguished. If the identifier of an imported interface record is omitted, the name of the interface type is used by default, i.e., *id* is equivalent to *id:id* in an IMPORTS list (see the discussion of DEFINITIONS FROM, however). The identifier following the colon must be defined in the directory. It can name an interface type, i.e., a DEFINITIONS module. Alternatively, it can name a PROGRAM module; this case is discussed later.

It is important to distinguish between interface types (declared in the DIRECTORY list) and interface records (declared in the IMPORTS list). Assume the following program skeleton:

```
DIRECTORY Defs1: FROM "defs1", Defs2: FROM "defs2";

Prog: PROGRAM IMPORTS Interface1: Defs1, Defs2 =
    BEGIN ... END.
```

Within the body of the program, *Interface1* refers to an interface record; *Defs1*, to an interface type. If *t* is a type or a constant, both *Interface1.t* and *Defs1.t* are valid, and they have identical meanings. On the other hand, if *proc* is an interface item, *Interface1.proc* names that item, but *Defs1.proc* is an error. The scope rules are arranged so that identifiers of the interface records introduced in the IMPORTS list are examined before those of the interface types introduced in the directory. Within the body of *Prog*, *Defs2* therefore refers to the interface record. Similar considerations apply to the use of the identifiers *Interface1*, *Defs1*, and *Defs2* with OPEN. Note that the distinction between the interface record and its type can be ignored unless the imported record is explicitly named.

The DEFINITIONS FROM construct does not mesh very well with IMPORTS and EXPORTS, but it is a well established feature of Mesa and the following convention has been adopted. Only an identifier introduced in the directory can appear in the list following DEFINITIONS FROM. If an interface record of that type is imported but is not given an explicit name in the IMPORTS list, then the record is opened; otherwise, the type. Thus the default naming convention results in record instances being opened. In the example above,

```
DEFINITIONS FROM Defs1, Defs2;
```

would open the type *Defs1* and the record *Defs2*. Again, confusion is possible only if the interface records are explicitly named.

The EXTERNAL attribute provided by previous versions of Mesa has been withdrawn. Externally defined procedures and signals must now be components of imported interfaces or be passed into modules as explicit parameters.

Importing and Exporting Program Modules

Each PROGRAM module exports itself, even if its module identifier is not mentioned in any interface. Any module can include a program module in its directory. A program module can import another by also mentioning that module in its own IMPORTS list. Since global frames cannot be embedded within other structures, the imported value in this case is a pointer to a global frame. The construct

```
... IMPORTS ... frame: Prog ...
```

has an effect similar to the declaration

```
frame: POINTER TO FRAME[Prog] = ...
```

where the initializing value is computed and assigned by the binder. The same default naming convention applies; if *frame* were omitted, *Prog* would be used as the identifier of the pointer variable.

Again, it is important to distinguish between the program constants declared in directory entries and the frame pointers supplied as imported values. Consider the following example:

```
DIRECTORY Prog1: FROM "prog1", Prog2: FROM "prog2";

Prog: PROGRAM IMPORTS frame1: Prog1, Prog2 =
    BEGIN ... END.
```

Within the body of *Prog*, *Prog1* names a program constant. The only legitimate uses for *Prog1* are to define a frame type (POINTER TO FRAME[*Prog1*]) and to create new instances of that type (NEW *Prog1*). The directory entry itself does not require the binder or loader to locate an existing instance of the program module. The appearance of *Prog1* (and *Prog2*) in the IMPORTS list does direct the binder to do this, however. Thus the value assigned to *frame1* points to a global frame that is already part of the configuration, and *frame1* can be used, e.g., to START that frame or to access its variables. By the default naming convention, the value of *Prog2* within the main body is a pointer that can be used only to refer to a frame, not to a program constant, i.e., the construct POINTER TO FRAME[*Prog2*] would be illegal.

> Even if *t* denotes a type or constant, it is no longer possible to use *Prog1.t* to name that value; *frame1.t* remains acceptable, however.
>
> By importing a program directly, a module gains access to all the public variables in that particular program. Importing a program as an interface element does not give such access but also does not couple the importer to a particular exporter. This is discussed further below.

Module Instantiation

The NEW operator creates new instances of program modules. It can be invoked explicitly; it is also invoked implicitly by the binder/loader in the course of creating a configuration. In either case, NEW causes no code within the instantiated module to be executed. Instead, START is now used to pass any arguments and to start execution of the main body (including initialization code).

The permissable operands of NEW are discussed below. NEW no longer permits an argument list, but the form

> NEW *id* [! ...]

is still available for catching signals associated with instantiation. The value of a NEW operation is always a pointer to the newly created frame.

Program modules now optionally return values. The allowed control disciplines depend somewhat upon whether a value is returned. Let *frame1* and *frame2* be pointers to frames of programs *Prog1* and *Prog2* respectively, and assume that *Prog1* returns a value but *Prog2* does not.

> The main body of *Prog1* must contain a RETURN statement. The first such RETURN terminates execution of that body but does not cause deallocation of the global frame. Thus frame pointers remain valid, procedures declared within instances of *Prog1* can be called and the like. The body of *Prog1* cannot contain any STOP statements.
>
> An expression with one of the forms
>
> START *frame1*[...]
> START *frame1*[... ! ...]
>
> is used to supply arguments to an instance of *Prog1* and to initiate its execution. The value of the expression is the value returned by *Prog1*. Execution of *Prog1* cannot be restarted after return.

The main body of *Prog2* cannot contain a RETURN statement, but it can contain any number of STOP statements. Execution of one of these suspends execution but causes no deallocation.

A statement with one of the forms

```
START frame2[...]
START frame2[... ! ...]
```

is used to supply arguments to an instance of *Prog2* and to initiate its execution. Execution of *Prog2* can be restarted by one of the statement forms

```
RESTART frame2
RESTART frame2[ ! ...]
```

Using alternating RESTARTs and STOPs, two cooperating programs can execute as SIMULA-like coroutines.

In all cases, the parameters of START are now type-checked, and there is no restriction on the number or size of such parameters.

Any attempt to invoke a procedure prior to starting the enclosing program's initialization code causes a *start trap.* If the program requires no parameters and returns no results, the system will attempt to START the global frame and then retry the procedure call. Thus explicit STARTs (and the corresponding proliferation of frame pointers) are not required in many common situations.

All imported interface variables are bound before the first START. On the other hand, a start trap occurs at most once for any given frame. Some care is therefore required when procedure calls during initialization can cycle through a set of frames; another procedure sharing the same global frame can be called before initialization is complete.

In the following example, assume that a start trap occurs because of an extramodular call of *ProcA2.* If *ProgramB* has not been started either, *ProcA1* will be executed before some components of *ProgramA*'s frame (such as *w*) are initialized.

```
ProgramA: PROGRAM IMPORTS DefsB EXPORTS DefsA =
  BEGIN
  v: T ← DefsB.procB1[...];
  w: INTEGER ← 0;
  ...
  ProcA1: PUBLIC PROCEDURE [...] RETURNS [...] = ...
  ...
  ProcA2: PUBLIC PROCEDURE [...] = ...
  ...
  END.

ProgramB: PROGRAM IMPORTS DefsA EXPORTS DefsB =
  BEGIN
  x: T ← DefsA.ProcA1[...];
  ...
  ProcB1: PUBLIC PROCEDURE [...] RETURNS [...] =
  ...
  END.
```

The implicit STOP that was previously inserted between a module's initialization code and its main body (if non-void) has been removed. Such a STOP must be specified explicitly.

Program Variables and NEW

Program types have been generalized. Their constructors have the following form:

```
ProgramTC ::= PROGRAM ParameterList ReturnsClause
```

and they can be used anywhere type expressions are legal, e.g., to define named types or to declare *program variables.* Values of the latter are pointers to global frames. Such variables allow type-correct manipulation of frame pointers without requiring commitment to a particular FRAME type. Both pointers to frames and program variables can be STARTed.

Imported interfaces can supply values for program variables as discussed above. In addition, the NEW operator creates such values dynamically. The domain of NEW has been extended to encompass the following two cases:

Program Constants

The identifier of the program module itself or of a directory entry denotes a *program constant.* When NEW is applied to such a value, the loader is invoked with a file name as an argument. The file name is taken from the directory or, in the case of the program's own identifier, constructed by the compiler. A global frame is allocated, the frame is connected to the code in the designated file, and imported interfaces within that frame are filled from the exported interfaces of the running configuration. This operation involves a directory search, etc., and is relatively expensive.

> In the current implementation, the designated file can contain only one PROGRAM module. System procedures must be used to instantiate configurations consisting of several modules (see Mesa 3.0 System Update).

Program Variables and Frame Pointers

The value of a program or pointer variable is a pointer to an existing global frame. Application of NEW to such a value creates a new copy of that global frame. All interface records in the new frame are assigned copies of the corresponding records in the original frame, i.e., all bindings are inherited. Other variables within the frame are *not* copied. This operation is relatively inexpensive.

In either case, the new frame is uninitialized, except for embedded interface records, until it is STARTed (perhaps by a start trap).

Assume the declarations

```
DIRECTORY Prog: FROM "prog";  -- assumed parameterless

Module: PROGRAM IMPORTS p: Prog =
    BEGIN
    prog1, prog2: PROGRAM;
    frame: POINTER TO FRAME[Prog];
    ...
    END.
```

Then the valid assignments are summarized by the following list:

```
frame ← p;                  -- copies a pointer
frame ← NEW p;              -- creates a copy of the imported frame
frame ← NEW Prog;           -- creates and binds a new instance

prog1 ← prog2;              -- copies a pointer
prog1 ← NEW prog2;          -- creates a copy of the frame
prog1 ← NEW frame;          -- creates a copy of a frame with known type
prog1 ← NEW Prog;           -- creates and binds a new instance.
```

Note that NEW yields a program type or a pointer type as required by context.

The assignment *frame* ← *prog1* is illegal. Although the following assignments are sensible, the current implementation does not support or allow them:

```
prog1 ← frame;
prog1 ← Prog .
```

Program variables are useful for STARTing or replicating global frames when a client does *not* want to be coupled to the internal details of a particular implementation.

Example:

```
DIRECTORY ProgDefs: FROM "progdefs";
Prog: PROGRAM [n: CARDINAL] EXPORTS ProgDefs =
    BEGIN
    v: PUBLIC Thing;
    Proc: PUBLIC PROCEDURE = ...;
    ...
    END.

ProgDefs: DEFINITIONS =
    BEGIN
    Prog: PROGRAM [CARDINAL];
    ...
    Proc: PROCEDURE;
    ...
    END.
```

A client that declares *p:* POINTER TO FRAME[*Prog*] (or, equivalently, imports *p: Prog*) receives a pointer to the global frame of the particular implementer *Prog*. That client is free to access, e.g., *p.v*; on the other hand, recompilation of all such clients is necessary when *Prog* changes (even trivially). A client that imports *ProgDefs* cannot mention *ProgDefs.Prog.v*, nor can *Proc* be referenced as *ProgDefs.Prog.Proc*. The appropriate value of *Proc* could be imported from the same interface, as suggested here. Alternatively, *Prog* could be changed so that the START of *ProgDefs.Prog* returned a value providing access to *Proc*.

Importing *ProgDefs* instead of *Prog* decouples implementer and client. An internal change in the former does not require recompilation of the latter. Indeed, it is possible for several quite different and independent implementations all to export *ProgDefs*.

Other Language Changes

Packed Arrays

The attribute PACKED has been introduced to specify the packing of arrays. It can be used in the following type constructors:

```
ArrayTC ::= PACKED ARRAY IndexType OF TypeSpecification | ...
ArrayDescriptorTC ::= DESCRIPTOR FOR PACKED ARRAY OF TypeSpecification | ...
```

The idea is that, for a packed array, the compiler will choose the most compact representation that it is prepared to support. Currently, bytes and words are the only supported units of packing. Thus values of types that can be represented in 8 bits or less (e.g., BOOLEAN as well as CHARACTER) are packed into bytes; all others are packed into words or integral multiples thereof.

All the usual array operations, such as indexing, construction, assignment and comparison (for equality and inequality only), apply to packed arrays. Note the following:

> LENGTH applied to a packed array (or descriptor thereof) yields the number of elements, not the number of words.
>
> In the form DESCRIPTOR [*array*], the PACKED attribute is inherited from *array*. In the form DESCRIPTOR [*base, n, type*], the PACKED attribute is deduced from context. The second argument *n* always specifies the number of elements.

Overlaid Variant Records

In the declaration of a variant record type, the word OVERLAID can replace COMPUTED. Overlaid variants behave exactly as computed variants with the following extension: any field of a particular variant can be accessed without discrimination if the name of that field is unique with respect to both the common part and all other variants (including any overlaid variant subparts).

Overlaid records can be used to breach the type system, as can variant records with computed tags in general.

> Example:
>
> ```
> R: TYPE = RECORD [
>     common: INTEGER,
>     variant: SELECT OVERLAID Color FROM
>         red => [b: BOOLEAN, i: [0..10)],
>         blue => [i: INTEGER, c: CHARACTER],
>         ENDCASE];
> v: R
> ```
>
> Then *v.common*, *v.b*, and *v.c* are all legal expressions. The first is always meaningful, but the second or third only makes sense if the value of *v* is a *red R* or a *blue R* respectively. The expression *v.i* is ambiguous and disallowed.

If an overlaid variant record is OPENed, all uniquely named fields can be used without further qualification. Identifiers that do not name unique fields are visible but ambiguous; their use will be flagged, not ignored. On the other hand, opening an overlaid variant record using the WITH ... SELECT construct or declaring a discriminated variant (e.g., a *red R*) works as usual. Independently of uniqueness of naming, fields of the selected variant are accessible, and fields of other variants are not.

Empty Intervals

Empty intervals are now allowed in type declarations, notably in declarations of the index types of arrays. In the absence of genuine sequences, this change makes the simulation of them somewhat less painful. Note that the subrange, although empty, does establish the origin of the index set, e.g., [0..0) and [1..1) are not equivalent.

Example (see also the declaration of *StringBody* below):

```
ThingSequence: TYPE = MACHINE DEPENDENT RECORD [...,
    length: CARDINAL,
    value: ARRAY [0..0) OF Thing];          -- the last field

p: POINTER TO ThingSequence ← Alloc[SIZE[ThingSequence] + n*SIZE[Thing]];
p↑ ← [..., length: n, value: ]; p.value[i] ← ...
```

The compiler allows @*p.value* but flags any attempt to use a vacuous value itself, as in *q.value* ← *p.value*. Note that subscript bounds are being breached here, not the type system.

Predeclared Identifiers

Every Mesa program is compiled in an environment that in effect contains the following declarations:

```
BOOLEAN: TYPE = {FALSE, TRUE};
-- make TRUE and FALSE pervasive
TRUE: BOOLEAN = TRUE;
FALSE: BOOLEAN = FALSE;

STRING: TYPE = POINTER TO StringBody;
StringBody: TYPE = MACHINE DEPENDENT RECORD [
    length: CARDINAL,
    maxlength: --READONLY-- CARDINAL,
    text: PACKED ARRAY [0..0) OF CHARACTER];

UNWIND: ERROR = CODE;
```

STRING and BOOLEAN remain reserved words.

BOOLEAN can now be used as the tag type of a variant record (but no new forms of SELECT are provided). Note that FALSE < TRUE.

The record type *StringBody* is declared in the way suggested above for simulating a sequence. As before, *s*[*i*] abbreviates *s*↑.*text*[*i*], etc.; this abbreviation does not extend to other "faked" sequences. Subject to the caveats above, the declaration of *StringBody* provides a way of embedding string bodies within other data structures; pointers to such embedded records can be assigned to normal string variables without breaching the type system.

Note that descriptors for the text portion of a string should be created using the form DESCRIPTOR [@*s.text*, *s.maxlength*], not DESCRIPTOR [*s.text*] (which will yield an incorrect length).

Identifier Clashes

It is now an error to declare the same identifier in both the input and output records of a procedure (program, signal, etc.) type. It is also an error to declare a local variable of a procedure (or program) by reusing an identifier of an input or output parameter of that procedure.

Use of Type Expressions

Arbitrary type expressions are now allowed as the arguments of SIZE, FIRST and LAST and as the final arguments of LOOPHOLE and DESCRIPTOR. Previously, these arguments were required to be type identifiers.

Examples:

```
LOOPHOLE[p, POINTER TO R]
LOOPHOLE[frame, PROGRAM [n: INTEGER]]
SIZE[ARRAY IndexSet OF T]
```

Compiler Changes

An assortment of minor bugs have been fixed; as usual, the fixes are documented separately. The following changes are noteworthy.

Variant Record Layout

The internal layouts of variant records have been changed to eliminate uninitialized gaps following the common part and to solve some long-standing problems related to tag alignment in discriminated variants. As a consequence of the latter, some discriminated variants now occupy more storage. The amount of storage required for a variant record is governed by the following rules:

> If the minimum amount of storage required for every variant is a word or less, each variant is adjusted to occupy the same number of bits as the longest.
>
> Otherwise, each variant is adjusted to occupy the minimum number of words.

Both of these are changes. Some discriminated variants that fit into a single word now consume more bits. This can make a difference only when the discriminated variant is itself embedded within a record.

Examples.

```
R1: TYPE = RECORD [
    SELECT tag: * FROM
        red => [b: BOOLEAN],
        blue => [c: CHARACTER],
        ENDCASE];

R2: TYPE = RECORD [
    SELECT tag: * FROM
        null => NULL,
        yellow => [b: BOOLEAN],
        green => [i: INTEGER],
        ENDCASE]
```

The minimum storage required for fields of various types is given by the following table (changes are noted):

R1	9	
red R1	9	(formerly 2)
blue R1	9	
R2	32	
null R2	16	(formerly 2)
yellow R2	16	(formerly 3)
green R2	32	

Since uninitialized gaps have been eliminated, comparison of two fully discriminated variants is now allowed. The compiler also permits comparison of undiscriminated values if there is an explicit tag and if all variants have the same length.

Checking Machine Dependent Records

The compiler now checks the declarations of MACHINE DEPENDENT record types. The fields of such a record must specify a contiguous area of storage. Declarations in which gaps would appear between fields are flagged as errors. Packing is governed by the following rules:

Fields minimally requiring a word or less cannot cross a word boundary.

Fields requiring more than a word must begin on a word boundary and occupy an integral number of words.

The first field begins on a word boundary.

Records occupying more than a word must occupy an integral number of words. The lengths of variant MACHINE DEPENDENT records must additionally conform to the rules stated in the previous section.

Compiler Switches and Options

Users of the compiler now have several options available. Switches that select the options are embedded (according to FTP's conventions) within the list of files to be compiled. The following switches are currently provided:

Switch	*Option Controlled*
pause	Pausing after errors.
warnings	Generation of warning messages (see below).
xref	Generation of cross-reference data.

In addition,

command	Converts the preceding string to a switch name.

Within a list of files, the form `option/c` sets the corresponding option for successive files, and any unambiguous initial substring of the switch name can be used for `option`. The form `file/o` sets the options `o` for a single file only, and any sequence of unambiguous initial switch characters can be used for `o`. In either form, a "-" or "~" inverts the sense of an switch. Switches can appear in the command line or in responses to the compiler's prompt "Compile:".

The pause option conditionally causes the compiler to halt and request action from the user before proceeding. A pause occurs only if errors have been detected. This option is useful when input is taken from the command line; it gives the user a chance to acknowledge errors and to abort subsequent processing. If it is specified by a global switch (using `pause/c`), the conditional pause occurs only at the end of the entire sequence of compilations but is controlled by errors detected anywhere in that sequence. If this option is specified locally (using `file/p`), the compiler will pause if errors are detected in any file processed up to that point.

The cross-reference information for source file `Name.Mesa` is written onto the file `Name.XRJ` for post-processing as described in separate documentation.

Examples:

Pause if errors are detected in any definitions file; generate a cross reference data and suppress warnings for *prog2*:

```
defs1 defs2 defs3/p prog1 prog2/x-w prog3
```

Never pause:

```
-pause/c . . .
```

The default switch settings are equivalent to

```
pause/c warnings/c -xref/c
```

Warning Messages

The compiler now optionally generates warning mesages when it detects legal but suspicious usage. Currently, the following situations are reported:

A declared variable (but not a record field, input parameter, or return value) that is not public and is never referenced within the module containing its declaration. (See also *Identifier Clashes* above).

Comparisons such as $c < 0$, when c is a CARDINAL.

An initializing declaration that assigns the same non-NIL pointer or descriptor value to two or more variables (in particular, string variables).

Interfaces that are imported but not used.

Declared identifiers that are not public but appear in some exported interface.

Warning messages appear in the error log but do not abort compilation. Generation of these messages is controlled by the `w` switch. The default is to print warnings; `-w` inhibits this.

Source/Object Mapping

The resolution of the tables recording source-object correspondence has been increased to the level of statement boundaries. Previously, only the intersections of statement and source line boundaries were recorded. This change allows more precise placement of breakpoints.

Entry Point Limit

Certain internal formats have been changed in a way that limits the number of entry items within a single program module to 128. This number is the *maximum* of the number of procedure (or program) bodies and the number of signal (or error) codes. This limit is also built into the D0 and Dorado hardware.

In addition, the number of interface items declared within a single definitions module cannot exceed 128. This number is the *sum* of the number of procedure, program, signal and error declarations.

Module Compatibility

Module Formats

The format of object modules has been changed to contain the information required by the new binder. The output of the compiler is a degenerate configuration description which can be processed by either the binder or the loader. *This change requires the recompilation of all existing Mesa programs.* To emphasize this change, the extension used for object file names has been changed from XM to BCD (binary configuration description).

Interface Versions

All type checking has been moved into the compiler; the binder and loader simply match interfaces according to interface (file) name and version stamp. It is therefore essential that consistency of DEFINITIONS modules be maintained within configurations, even when such modules contain no type declarations.

Distribution

MesaUsers
MesaGroup